Add new user-song associations in batches, allowing a significant period of time between each batch.

Since the total that is in the denominator of all the p calculations will not change in between batches, that makes it possible, at the end of a batch load, to create a one-diminsional array to represent the p log p values, where the index is the numerator in the p calculation. Thus, each relevant p log p calculation only needs to be performed once, and is then reused.

Instead of actually re-allocating memory for the array at the end of each batch load, the array can be zeroed out. A 0 in an element indicates that p log p has not yet been calculated. So, when a value is needed for p log p, the appropriate element is checked, and if it is 0, it is calculated. If it is non-zero, then the value that is there is used.

Appendix C

```
#VERSION 12   08/27/00

#Copyright (c) 2000 by Virtual Development Corp. All Rights Reserved.

#Usage Notes##########################
# MimimumConvergenceIterations in the Config file must be at least 1. (See BUGS.)

# MinimumConvergenceIterations "beats" MaxTime. It will run for the minimum
#   number of configurations, then run until MaxTime.



# work_ = Work instance
# rel_  = Relatable instance
# clus_ = Cluster
# clst_ = ClusterSet
# clss_ = ClusterSetSignature

import whrandom
import math
import xmllib
import copy
import time
import ConfigParser
import urllib
import sys



# Utility stuff
G_generator = whrandom.whrandom()  # For why global, see
http://starship.python.net/crew/donp/script/sample.py
#G_generator.seed(1,1,3)

def shuffle(sample_size):    # See
http://starship.python.net/crew/donp/script/sample.py
    '''Moses and Oakford algorithm.  See Knuth, vol 2, section 3.4.2.
    Returns a random permutation of the integers from 1 to
    sample_size.
    '''
    assert type(sample_size) == type(0) and sample_size > 0
    global G_generator
    list = range(1, sample_size + 1)
    for ix in xrange(sample_size - 1, 0, -1):
        rand_int = G_generator.randint(0, ix)
        if rand_int == ix:
            continue
        tmp = list[ix]
        list[ix] = list[rand_int]
        list[rand_int] = tmp
    return list

# from http://starship.python.net/pipermail/python-de/1997q1/000026.html
#"Converter module from strings to HTML entities"
# The code is modified slightly modified to use the encodings
# the python xml parser defaults to decoding, rather than using
# htmlentitydefs.

EntitiesByOrd={ ord('<')   : 'lt',
   ord('>')  : 'gt',
   ord('&') : 'amp',
   ord('"') : 'quot',
   ord("'") : 'apos' }

def toXML(s):
  pos=start=0
  result=""
  flush=0
  while pos<len(s):
    c=ord(s[pos])
    if EntitiesByOrd.has_key(c):
      flush=1
      item="&"+EntitiesByOrd[c]+";"
    if flush:
      result=result+s[start:pos]+item
      start=pos+1
      flush=0
    pos=pos+1
  result=result+s[start:pos]
  return result

def computeEvenRankUnitRanks( lstTup_input ):
  # SHOULD BE IN DATA object

  # Suppose 100 values are tied for second place, and 1
  # is alone in first. It should not be assumed that we
  # should put the lone value in the top percentile, because
  # it could easily be due to noise. So, we compromise by
  # saying there are 2 ranks, and we assign .25 to everyone in the low
  # and .75 to the one in the high.

  # We only use the first element in the tuple for ranking.

  # Output list has the same data as the input, but in
  # rank order, and each tuple has two extra elements
  # at the end: the integer rank (ties are counted as
  # the same rank; best is highest) and the unit rank.

  # FURTHER ADJUSTMENT DURING TIME OF LITTLE DATA!!!! If
  # there are two input sort field values, 1 and 2, the
  # original algorithm gives outputs .25 and .75. But that
  # still means that the low level is much closer to 0
  # than the high level is. That makes no sense.
  # So, we change the levels to .625 and .875.


  lstTup_input.sort()
  assert lstTup_input[ 0 ][ 0 ] != None   # logic assumes first sort value is not None
  lstTup_intermediate = []
  int_rank = 0
  any_previousSortValue = None
  for tup_ in lstTup_input:
    if any_previousSortValue != tup_[ 0 ]:
      int_rank = int_rank + 1
      any_previousSortValue = tup_[ 0 ]
    lstTup_intermediate.append( tup_ + ( int_rank, ))

  lstTup_output = []
```

```
 for tup_ in lstTup_intermediate:
  float_ = ( tup_[ -1 ] - 0.5 ) / float( int_rank )
  float_tuning = Config.float_tuningRankBottom + float_ * ( 1.0 -
Config.float_tuningRankBottom )  #see note above for little data
  lstTup_output.append( tup_ + (float_tuning,) )


 return lstTup_output


def computeAverageUnitRanks( lstTup_input ):
 # NOT USED IN CURRENT CODE 8/24/00
 # The first element in the tuple is the only one used
 # in the ranking.
 # The output list contains tuples identical to the input
 # list but with an added element at the end, which is
 # the ranking, with dups assigned to the average ranks
 # of the dups.

 def isLastInDupSet( int_index, lstTup_ ):
  if len( lstTup_ ) == int_index + 1:
   return 1
  else:
   if lstTup_[ int_index ][ 0 ] != lstTup_[ int_index + 1 ][ 0 ]:
    return 1
   else:
    return 0

 float_offset = 1.0 / ( 2.0 * len( lstTup_input ))
 lstTup_input.sort()
 lstTup_output = []
 int_startDupIndex = 0
 int_limitIndex = len( lstTup_input )
 lst_currentDupSet = []
 for int_index in range( int_limitIndex ):
  if isLastInDupSet( int_index, lstTup_input ):
   lst_currentDupSet.append( lstTup_input[ int_index ] )

   # Compute average unit rank
   float_averageRank = ( int_index + int_startDupIndex ) / 2.0
   float_averageUnitRank = float_offset + float_averageRank / int_limitIndex

   # Add to output list
   for tup_ in lst_currentDupSet:
    lstTup_output.append( tup_ + ( float_averageUnitRank, ))

   # Set the stage for next iteration
   int_startDupIndex = int_index + 1
```

```
      lst_currentDupSet = []
    else:
      lst_currentDupSet.append( lstTup_input[ int_index ])

  return lstTup_output



# Classes

class Config:
  # When an instance is created, the class attributes are populated;
  # at that point, the instance itself can be thrown away.

  str_clusterFile = None
  str_useFile = None
  str_oldUseFile = None
  int_createClusterCount = None
  float_maxTime = None
  int_minimumConvergenceIterations = None
  str_outClusterFile = None
  float_tuningRankBottom = None
  float_tuningZeroWeight = None

  C_str_configFile = 'clusterconfig.txt'
  C_str_sectionName = 'Configuration'
  C_str_clusterFile = 'InClusterFile'
  C_str_useFile = 'UseFile'
  C_str_oldUseFile = 'OldUseFile'
  C_str_createClusterCount = 'CreateClusterCount'
  C_str_maxTime = 'MaxTime'
  C_str_minimumConvergenceIterations = "MinimumConvergenceIterations"
  C_str_outClusterFile = 'OutClusterFile'
  C_str_tuningRankBottom = 'TuningRankBottom'
  C_str_tuningZeroWeight = 'TuningZeroWeight'

  def __init__( self ):
    configParser = ConfigParser.ConfigParser()
    configParser.read( Config.C_str_configFile )
    Config.str_clusterFile  = configParser.get( Config.C_str_sectionName,
Config.C_str_clusterFile )
    Config.str_useFile  = configParser.get( Config.C_str_sectionName,
Config.C_str_useFile )
    Config.str_oldUseFile  = configParser.get( Config.C_str_sectionName,
Config.C_str_oldUseFile )
    Config.int_createClusterCount  = int( configParser.get( Config.C_str_sectionName,
Config.C_str_createClusterCount ))
```

```
    Config.float_maxTime = float( configParser.get( Config.C_str_sectionName,
Config.C_str_maxTime ))
    Config.int_minimumConvergenceIterations = int( configParser.get(
Config.C_str_sectionName, Config.C_str_minimumConvergenceIterations ))
    Config.float_tuningRankBottom  = float( configParser.get( Config.C_str_sectionName,
Config.C_str_tuningRankBottom ))
    Config.float_tuningZeroWeight  = float( configParser.get( Config.C_str_sectionName,
Config.C_str_tuningZeroWeight ))
    Config.str_outClusterFile  = configParser.get( Config.C_str_sectionName,
Config.C_str_outClusterFile )


class Data:
  # This is a singleton. One instance is created, and that creates everything.

  # "Longnames" are of the format "Beatles - Hey Jude". The artist and the title
separate by
  # spacedashspace. Each Work object is uniquely identified by a Longname.

  singleton = None

  def __init__( self ):
    assert not self.__class__.singleton
    self.__class__.singleton                    = self
    self.dictStrDictStrNone_userLongname = {}
    self.dictStrDictStrFloat_longname2Longname1UnitRank      = {}
    self.dictLongnameWork_ = {}
    self.dictStrDictStrInt_longname1Longname2Count = {}
    self.dictStrInt_longnameUniqueCount = {}
    self.lstWork_ = []

    assert Config.str_useFile
    print 'about to read data'
    self.__readUserPlayStats( Config.str_useFile )
    print 'about to generate use counts'
    self.__generateUseCounts()
    print 'about to generate unit ranks'
    self.__generateUnitRanks()

  def displayCheckingInfo( self ):
    dict_russians = self.dictStrDictStrFloat_longname2Longname1UnitRank[ 'Sting -
Russians' ]
    lst_russians = dict_russians.items()
    lst_russians.sort()



  def getWorks( self ):
    return self.lstWork_

  def getUnitRanks( self ):
    assert self.dictStrDictStrFloat_longname2Longname1UnitRank
    return self.dictStrDictStrFloat_longname2Longname1UnitRank

  def getAssociatedLongnames( self, str_longname ):
    assert self.dictStrDictStrFloat_longname2Longname1UnitRank.has_key( str_longname )
    return self.dictStrDictStrFloat_longname2Longname1UnitRank[ str_longname ].keys()


  def __readUserPlayStats( self, str_fileName ):
    if str_fileName[ :7 ] == "http://":
      fil_ = urllib.urlopen(str_fileName)
    else:
      fil_ = open(str_fileName,'r')
    str_ = fil_.read()
    fil_.close()

    class UseListContainerParser1( xmllib.XMLParser ): # Embedded class, only used
here!
      # THIS LOGIC ASSUMES UNIQUENESS AT USER/SONG LEVEL IN THE INPUT XML FILE!!

      def __init__( self, data_ ):
        self.str_currentUser = None
        self.data_ = data_
        xmllib.XMLParser.__init__( self )

      def start_entry( self, dict_ ):
        # str_work is the title of the work, which must be distinguished from Work
objects!

        if ( self.str_currentUser != 'mike3k@mail.com'
            and self.str_currentUser != 'jake@jspace.org'
            and self.str_currentUser != 'jake@braincase.net' ):
          if int( dict_[ 'count' ] ) > 1:
            str_artist = intern( dict_['artist'] )
            str_work = intern( dict_[ 'work' ] )
            str_longname = intern( '%s - %s' % ( str_artist, str_work ))

            dict_ = self.data_.dictStrInt_longnameUniqueCount
            if dict_.has_key( str_longname ):
              dict_[ str_longname ] = dict_[ str_longname ] + 1
            else:
              dict_[ str_longname ] = 1


      def start_useList( self, dict_ ):
        self.str_currentUser = dict_[ 'user' ]

    class UseListContainerParser2( xmllib.XMLParser ): # Embedded class, only used
here!

      def __init__( self, data_ ):
        self.str_currentUser = None
        self.data_ = data_
        xmllib.XMLParser.__init__( self )

      def start_entry( self, dict_ ):
        # str_work is the title of the work, which must be distinguished from Work
objects!
        str_artist = intern( dict_['artist'] )
        str_work = intern( dict_[ 'work' ] )
        str_longname = intern( '%s - %s' % ( str_artist, str_work ))

        if ( self.data_.dictStrInt_longnameUniqueCount.has_key( str_longname ) and
            self.data_.dictStrInt_longnameUniqueCount[ str_longname ] > 1 ):
          if self.data_.dictStrDictStrNone_userLongname.has_key( self.str_currentUser ):
            if self.data_.dictStrDictStrNone_userLongname[ self.str_currentUser
].has_key( str_longname ):
              pass     # Already there!
            else:
              self.data_.dictStrDictStrNone_userLongname[ self.str_currentUser ][
str_longname ] = None
          else:
            self.data_.dictStrDictStrNone_userLongname[ self.str_currentUser ] = {
str_longname : None }

          if not self.data_.dictLongnameWork_.has_key( str_longname ):
            work_ = Work( str_longname, str_artist, str_work )
            self.data_.lstWork_.append( work_ )
            self.data_.dictLongnameWork_[ str_longname ] = work_

      def start_useList( self, dict_ ):
        self.str_currentUser = dict_[ 'user' ]


    parser_1 = UseListContainerParser1( self )
    parser_1.feed( str_ )
    parser_1.close()
    parser_2 = UseListContainerParser2( self )
    parser_2.feed( str_ )
    parser_2.close()

  def __generateUseCounts( self ):
    dictStrDictStrInt_longname1Longname2Count = {}
  lstStr_user =self.dictStrDictStrNone_userLongname.keys()
  int_loopCount = 0
  for str_user in lstStr_user:
   int_loopCount = int_loopCount + 1
   int_innerLoopCount = 0
   sys.stdout.flush()
   for str_longname1 in self.dictStrDictStrNone_userLongname[ str_user ].keys():
    int_innerLoopCount = int_innerLoopCount + 1
#   print 'deep in loop, ', int_innerLoopCount, ' of ',
len(self.dictStrDictStrNone_userLongname[ str_user ])
    for str_longname2 in  self.dictStrDictStrNone_userLongname[ str_user ].keys():
#     if str_longname1 != str_longname2: songs played by only 1 user can still be
clustered due
#                               to the user's other choices...
not counting cases
#                               where the two are equal would
eliminate them, and
#                               should cause logic that loops
through all of the songs
#                               looking for unitRanks to fail
     if str_longname1 != str_longname2:
      if dictStrDictStrInt_longname1Longname2Count.has_key( str_longname1 ):
       if dictStrDictStrInt_longname1Longname2Count[ str_longname1 ].has_key(
str_longname2 ):
        dictStrDictStrInt_longname1Longname2Count[ str_longname1 ][ str_longname2
] = \
            dictStrDictStrInt_longname1Longname2Count[ str_longname1 ][
str_longname2 ] + 1
       else:
        dictStrDictStrInt_longname1Longname2Count[ str_longname1 ][ str_longname2
] = 1
      else:
       dictStrDictStrInt_longname1Longname2Count[ str_longname1 ] = {
str_longname2 : 1 }
  self.dictStrDictStrInt_longname1Longname2Count =
dictStrDictStrInt_longname1Longname2Count


 def __generateUnitRanks( self ):
  # "Unit ranks" are ranks scaled down to the unit interval. For instance, the lowest
  # rank out of 57 elements is 0, and the highest is  56/57 = .98245614035. But, we
  # also perform averaging, so ranks that extreme should be unusual.

  # Consider longname1 to be a work "associated" with longname2. Longname2 is the
work
      # for which we are generating a profile; this profile involves the
associated
  # Longname1 works.
```

```
    # That is, a profile for a longname2 would contain all
    # the longname1's that are associated with it. For each associated work, considered
across all
    # main works, there is one rank for each main work,
    # that's where the uniform distribution comes from. The alternative would be: for
each main work have
    # one rank for each associated work; then some associated works would NECESSARILY
have very low rank.
    # In contrast, using the approach presented, all associated works CAN have high
rank -- but under
    # the null hypothesis the distribution would be uniform.



    self.dictStrDictStrFloat_longname2Longname1UnitRank      = {}
    for str_longname1 in self.dictStrDictStrInt_longname1Longname2Count.keys():
      lstTupIntStr_ = []
      dictStrInt_longname2Count = self.dictStrDictStrInt_longname1Longname2Count[
str_longname1 ]
      for str_longname2 in dictStrInt_longname2Count.keys():
        lstTupIntStr_.append(( dictStrInt_longname2Count[ str_longname2 ], str_longname2
))
      if str_longname1 == 'Elton John - Levon':
        lstTupIntStr_.sort()
      lstTupIntStrIntFloat_ = computeEvenRankUnitRanks( lstTupIntStr_ )
      for int_ in range( len( lstTupIntStrIntFloat_ )):
        tupIntStrIntFloat_ = lstTupIntStrIntFloat_[ int_ ]
        float_ = tupIntStrIntFloat_[ -1 ]
        str_longname2 = lstTupIntStrIntFloat_[ int_ ][ 1 ]
        if self.dictStrDictStrFloat_longname2Longname1UnitRank.has_key( str_longname2 ):
          self.dictStrDictStrFloat_longname2Longname1UnitRank[ str_longname2 ][
str_longname1 ] = float_
        else:
          self.dictStrDictStrFloat_longname2Longname1UnitRank[ str_longname2 ] = {
str_longname1 : float_ }

#   fil_.close()

# computeAverageUnitRanks

class Relatable:
  def getName( self ):
    assert 0

  def getAssociatedRelatedness( self, str_otherName ):
    assert 0

  def getAssociatedLongnames( self ):
    assert 0
  def getOverallRelatedness( self, rel_ ):

    float_zeroWeight = Config.float_tuningZeroWeight

    float_sum = 0.0
    float_divisor = 0.0
    for str_name in self.getAssociatedLongnames():
      float_other = rel_.getAssociatedRelatedness( str_name )
      if float_other == None:   # Defensive programming
        float_other = 0.0
      if float_other == 0:
        float_weight = float_zeroWeight
      else:
        float_weight = 1.0
      float_divisor = float_divisor + float_weight
      float_self = float( self.getAssociatedRelatedness( str_name )) # Cast is
defensive programming
      float_product = float_self * float_other * float_weight
      float_sum = float_sum + float_product
    if float_divisor:
      float_overallRelatedness = float_sum / float_divisor
    else:
      float_overallRelatedness = 0.0
    return float_overallRelatedness

class Work( Relatable ):
  # The xml attribute 'work' is the title of the work, which must be distinguished from
Work objects,
  # which contain artist info as well as title info!

  def __init__( self, str_longname, str_artist, str_work ):
    # The "Longname" of the work, for purposes of this program, is the artist + the
work title.
Data.singleton.getAssociatedLongnames( str_longname )
    self.str_longname = str_longname
    self.str_artist = str_artist
    self.str_work = str_work

  def getName( self ):
    return self.str_longname

  def getArtist( self ):
    return self.str_artist

  def getAssociatedRelatedness( self, str_longname ):
    dictStrDictStrFloat_longname2Longname1UnitRank = Data.singleton.getUnitRanks()
  dict_ = dictStrDictStrFloat_longname2Longname1UnitRank   #Using intermediate name
just for clarity
  assert dict_.has_key( self.str_longname )
  if dict_[ self.str_longname ].has_key( str_longname ):
   float_unitRank = dict_[ self.str_longname ][ str_longname ]
  else:
   float_unitRank = 0.0
  return float_unitRank

 def getAssociatedLongnames( self ):
  return Data.singleton.getAssociatedLongnames( str_longname )

class Cluster( Relatable ):
 # To understand this class, it's important to understand the difference between a
 # cluster's membership list and its profile. Both of them involve a group of
 # objects subclassed from Relatable. But the membership list (self.lstRel_member)
 # determines the objects that are currently members of a cluster; whereas, the
 # profile (self.dictStrFloat_longnameRelatedness) is a description of the current
 # "center" of the cluster for purposes of measuring the distance between the
 # cluster and an object that is a candidate for membership in the cluster.

 # Normally, all candidate objects are assigned to a cluster before the profile
 # is computed; these assignments are based on the old profiles. For instance,
 # when clusters are being generated for the first time, the old profiles are
 # random. When clusters are being regenerated based on old clusters read from
 # an xml disk file, the profiles from the disk file clusters are used as the
 # old profiles.

 str_nextAutomaticName = '1'

 def __init__( self, str_name=None ):
  self.lstRel_member = []
  self.dictStrFloat_longnameRelatedness = {}
  if str_name:
   self.str_name = str_name
  else:
   int_ = int( self.__class__.str_nextAutomaticName )
   self.str_name = self.__class__.str_nextAutomaticName
   self.__class__.str_nextAutomaticName = str( int_ + 1 )

 def getName( self ):
  return self.str_name

 def getMembers( self ):
  return self.lstRel_member

 def getAssociatedRelatedness( self, str_longname ):
  # 1 or 0
  if self.dictStrFloat_longnameRelatedness.has_key( str_longname ):
    return  self.dictStrFloat_longnameRelatedness[ str_longname ]
  else:
    return 0.0

def getCountUniqueArtist( self ):
  if not self.lstRel_member:
    return 0
  assert self.lstRel_member[ 0 ].__class__ == Work
  dict_ = {}
  for work_ in self.lstRel_member:
    dict_[ work_.getArtist() ] = None
  return len( dict_ )

def getAssociatedLongnames( self ):
  return self.dictStrFloat_longnameRelatedness.keys()

def addToCluster( self, rel_ ):
  self.lstRel_member.append( rel_ )

def addToProfile( self, strLongname ):
  # Used for initializing empty profile for later clustering.
  self.dictStrFloat_longnameRelatedness[ strLongname ] = None

def computeClusterProfile( self, bool_binary ):
  # Normally, relatedness of each member to the cluster is binary --
  # 1 if it's in the dict, 0 otherwise. However, in the final
  # cluster confergence, it makes sense to do a 2-stage profile computation;
  # first we compute the binary values (represented by membership in
  # the dict vs. non-membership), then, using those values, we recompute
  # the profile, generating floating point values. This allows
  # us, in the final convergence, to generage clusters in such
  # a way that the most remote profile elements don't hold as great a sway
  # over what potential members are attracted to the cluster.


  # WHILE REVIEWING THIS CODE FOR WORK ON CLUSTERS13, I NOTICED THAT THIS
  # APPARENTLY SHOULD BE STRUCTURED AS: IF BOOL_BINARY...ELSE. THIS WOULD
  # AVOID SETTING dictStrFloat_longnameRelatedness TWICE, AS APPARENTLY
  # HAPPENS WITH THE CURRENT CODE. NOT CHANGING NOW BECAUSE AM WORKING
  # ON NEW VERSION AND DO NOT EXPECT TO TEST CHANGES.

  for rel_ in self.lstRel_member:
    if rel_.__class__ == Work:
      self.dictStrFloat_longnameRelatedness[ rel_.getName() ] = 1.0
    elif rel_.__class__ == Cluster:
      lstStr_otherName = rel_.getAssociatedLongnames()
      for str_otherName in lstStr_otherName:
```

```
     self.dictStrFloat_longnameRelatedness[ str_otherName ] = 1.0
   else:
    assert 0  # Attempt to cluster an illegal class

  if not bool_binary:
   for rel_ in self.lstRel_member:
    if rel_.__class__ == Work:
     self.dictStrFloat_longnameRelatedness[ rel_.getName() ] =
self.getOverallRelatedness( rel_ )
    elif rel_.__class__ == Cluster:
     lstStr_otherName = rel_.getAssociatedLongnames()
     for str_otherName in lstStr_otherName:
      self.dictStrFloat_longnameRelatedness[ str_otherName ] =
self.getOverallRelatedness( rel_ )
    else:
     assert 0  # Attempt to cluster an illegal class


 def makeEmpty( self ):
  # Notice that it leaves the profile (self.dictStrFloat_longnameRelatedness) intact
for purposes
  # of getAssociatedRelatedness() and getAssociatedLongnames().

  self.lstRel_member = []

 def merge( self ):
  # Turns a cluster of clusters (each of which must contain works)
  # into a cluster of works

  lstWork_ = []

  for clus_ in self.lstRel_member:
   assert clus_.__class__ == Cluster
   for work_ in clus_.getMembers():
    assert work_.__class__ == Work
    lstWork_.append( work_ )
  self.lstRel_member = lstWork_

class ClusterSet:
 def __init__( self, str_fileName=None, lstClus_persistent=None,
int_randomClusterCount=None ):
  # The constructor just loads or creates the clusters, it doesn't
  # do any processing.
  # When constructing from a file, the clusters
  # have profiles for measuring relatedness, but have no members.
  # When constructing from a list of clusters, they keep their members.
  # Randomly generated clusters are given members.
  self.lstClus_ = []
```

```
    if str_fileName:
      __readUserPlayStats( str_fileName )

    elif int_randomClusterCount:
      lstWork_ = Data.singleton.getWorks()
      int_countWorks = len( lstWork_ )
      lstInt_shuffled = shuffle( int_countWorks )
      if int_countWorks < int_randomClusterCount: # Obviously only applicable in small
tests.
        int_randomClusterCount = int_countWorks
      int_numberOfRandomWorksPerCluster = int_countWorks / int_randomClusterCount
      clus_current = None
      for int_ in xrange( int_countWorks ):
        if int_ % int_numberOfRandomWorksPerCluster == 0:
          if clus_current:           #Skip first iteration
            clus_current.computeClusterProfile( bool_binary=1 )
          clus_current = Cluster()
          self.addToClusterSet( clus_current )
        clus_current.addToCluster( lstWork_[ lstInt_shuffled[ int_ ] - 1 ] )
      clus_current.computeClusterProfile( bool_binary=1 )  # May end up doing this
twice for a cluster
    else:
      assert lstClus_persistent
      self.lstClus_ = lstClus_

  def consolidateArtists( self ):
    # Move all works for a given artist to the cluster with the greatest
    #   concentration of works for that artist.

    # This may not be necessary in implementations where can do all clustering at
artist level.

    dictStrDictClusInt_artistClusterCount = {}
    dict_ = dictStrDictClusInt_artistClusterCount  # short handle

    for clus_ in self.lstClus_:
      lstWork_ = clus_.getMembers()
      for work_ in lstWork_:
        str_artist = work_.getArtist()
        if dict_.has_key( str_artist ):
          if dict_[ str_artist ].has_key( clus_ ):
            dict_[str_artist ][ clus_ ] = dict_[ str_artist ][ clus_ ] + 1
          else:
            dict_[ str_artist ][ clus_ ] = 1
        else:
          dict_[ str_artist ] = { clus_ : 1 }
    dictStrClus_artistBestCluster = {}

    for str_artist in dict_.keys():
      clus_bestCluster = None
      int_bestCount = 0
      for tupClusInt_ in dict_[ str_artist ].items():
        if tupClusInt_[ 1 ] > int_bestCount:
          int_bestCount = tupClusInt_[ 1 ]
          clus_bestCluster = tupClusInt_[ 0 ]
      dictStrClus_artistBestCluster[ str_artist ] = clus_bestCluster

    for clus_ in self.lstClus_:
      clus_.makeEmpty()

    dictStrLstWork_artistWork = {}
    for work_ in Data.singleton.getWorks():
      str_artist = work_.getArtist()
      if dictStrLstWork_artistWork.has_key( str_artist ):
        dictStrLstWork_artistWork[ str_artist ].append( work_ )
      else:
        dictStrLstWork_artistWork[ str_artist ] = [ work_ ]

    for tupStrClus_ in dictStrClus_artistBestCluster.items():
      str_artist = tupStrClus_[ 0 ]
      clus_ = tupStrClus_[ 1 ]
      for work_ in dictStrLstWork_artistWork[ str_artist ]:
        clus_.addToCluster( work_ )

    for clus_ in self.lstClus_:
      clus_.computeClusterProfile()

  def getAverageSquaredUniqueArtist( self ):
    int_sum = 0
    for clus_ in self.lstClus_:
      int_count = clus_.getCountUniqueArtist()
      int_sum = int_sum + int_count**2.0

    return float( int_sum ) / len( self.lstClus_ )

  def getAverageCountUniqueArtist( self ):
    int_sum = 0
    for clus_ in self.lstClus_:
      int_count = clus_.getCountUniqueArtist()
      int_sum = int_sum + int_count

    return float( int_sum ) / len( self.lstClus_ )
  def getMaxCountUniqueArtist( self ):
    int_max = 0
    for clus_ in self.lstClus_:
      int_count = clus_.getCountUniqueArtist()
      if int_count > int_max:
        int_max = int_count
    return int_max

  def getMinCountUniqueArtist( self ):
    int_min = len( Data.singleton.getWorks())
    for clus_ in self.lstClus_:
      int_count = clus_.getCountUniqueArtist()
      if int_count < int_min:
        int_min = int_count
    return int_min

  def getMaxClusterSize( self ):
    int_maxSize = 0

    for clus_ in self.lstClus_:
      int_size = len( clus_.getMembers())
      if int_size > int_maxSize:
        int_maxSize = int_size

    return int_maxSize


  def getSignature( self ):
    # Returns a dictionary which is a signature of the cluster
    # Convenient since dicts can be tested for equality, don't need identity
    dictStrDictStrNone_longnameLongname = {}
    for clus_ in self.lstClus_:
      str_clusterName = clus_.getName()
      dictStrDictStrNone_longnameLongname[ str_clusterName ] = {}
      for str_associatedLongname in clus_.getAssociatedLongnames():
        dictStrDictStrNone_longnameLongname[ str_clusterName ][ str_associatedLongname ]
= None
    return dictStrDictStrNone_longnameLongname


  def performClustering( self, lstRel_item, bool_recluster=0, bool_binary=1 ):
    # bool_recluster means recluster items that are already clustered.

    # For defensive programming, we copy the list object (nothing in the list is
copied)
    # so that, when we add to the list below, it doesn't have side effects
    # for calling methods which expect the list to be unmodified
    lstRel_itemToCluster = copy.copy( lstRel_item )

    if bool_recluster:
      for clus_ in self.lstClus_:
        for rel_ in clus_.getMembers():
          lstRel_itemToCluster.append( rel_ )

    for clus_ in self.lstClus_:
      clus_.makeEmpty()                   # Leaves profile intact
    for rel_ in lstRel_itemToCluster:
      float_bestRelatedness = 0.0              # default to no correlation
      clus_best = None
      for clus_ in self.lstClus_:
        float_currentRelatedness = clus_.getOverallRelatedness( rel_ )
        if float_currentRelatedness > float_bestRelatedness:
          float_bestRelatedness = float_currentRelatedness
          clus_best = clus_
      if float_bestRelatedness:     # IF 0 DOES NOT GO INTO A CLUSTER!!
        clus_best.addToCluster( rel_ )

      clus_.computeClusterProfile( bool_binary )             # Prepare the cluster
center for use in further correlation


  def convergeClusters( self, float_latestTime, int_minimumIterations, bool_binary=1 ):
    # float_latestTime is latest time to start an iteration

    float_currentTime = time.time()
    dict_oldSignature = None
    int_iterations = 0
    bool_done = 0
    while not bool_done:
      if int_iterations < int_minimumIterations or float_currentTime <=
float_latestTime:
        print 'iterating:', int_iterations
        self.performClustering( [], bool_recluster=1, bool_binary=bool_binary )
        dict_newSignature = self.getSignature()
        if dict_newSignature == dict_oldSignature:
          print 'finishing convergence due to unchanged signatures'
          bool_done = 1
        else:
          dict_oldSignature = dict_newSignature
          float_currentTime = time.time()
          int_iterations = int_iterations + 1
      else:
        print 'finishing due to timeout'
        bool_done = 1
  def merge( self ):

    for clus_ in self.lstClus_:
      clus_.merge()

  def getClusters( self ):
    return self.lstClus_

  def addToClusterSet( self, clus_ ):
    self.lstClus_.append( clus_ )

  def __readUserPlayStats( self, str_fileName ):
    # We do not put members into the clusters, we only populate the profiles.
    self.lstClus_ = []
    fil_ = open(str_fileName, 'r')
    str_ = fil_.read()
    fil_.close()

    class ClusterParser( xmllib.XMLParser ): # Embedded class, only used here!

      def __init__( self, clst_ ):
        self.clst_ = clst_
        self.clus_current = None
        xmllib.XMLParser.__init__( self )

      def start_member( self, dict_ ):
        str_artist = intern( dict_['artist'] )
        str_title = intern( dict_[ 'work' ] )
        tupStrStr_artistTitle = ( str_artist, str_title )
        str_longname = intern( '%s - %s' % tupStrStr_artistTitle )
        self.clus_current.addToProfile( str_longname )

      def start_cluster( self, dict_ ):
        self.clus_current = Cluster( dict_[ 'name' ] )
        clst_.lstClus_.append( clus_current )

    parser_ = ClusterParser( self )
    parser_.feed( str_ )
    parser_.close()

  def writeToDisk( self, str_fileName ):
    fil_ = open( str_fileName, 'w' )
    fil_.write( '<?xml version="1.0" encoding="ISO-8859-1"?>\n' )
    fil_.write( """<ClusterContainer xmlns:xsi="http://www.w3.org/1999/XMLSchema-
instance"
          xsi:noNamespaceSchemaLocation='ViewListContainer.xsd'>\n""" )
    fil_.write( '    <clusters medium="music">\n' )
  for clus_ in self.lstClus_:
   fil_.write( '     <cluster name="%s">\n' % clus_.getName())
   lstTup_ = []
   for work_ in clus_.getMembers():
    float_relatedness = clus_.getOverallRelatedness( work_ )
    tup_ = ( float_relatedness, toXML( work_.str_artist ), toXML( work_.str_work ))
    lstTup_.append( tup_ )
   lstTup_.sort()
   lstTup_.reverse()
   for tup_ in lstTup_:
    fil_.write( '        <member artist="%s" work="%s" relatedness="%s" />\n' %
( tup_[ 1 ], tup_[ 2 ], tup_[ 0 ]))
   fil_.write( '     </cluster>\n' )
  fil_.write( '   </clusters>\n' )
  fil_.write( '</ClusterContainer>\n' )
  fil_.close()



#####################################################################################
########

# SCRIPT LOGIC

try:

 Config()   # Get configuration data

 Data()    # Create data singleton


 if Config.int_createClusterCount:
  # See http://www.math.tau.ac.il/~nin/learn98/idomil/
  int_numberOfClusters = int( Config.int_createClusterCount * math.log(
Config.int_createClusterCount ))
  float_maxTime = time.time() + Config.float_maxTime
  float_mostFabulous = float( len( Data.singleton.getWorks()) * len(
Data.singleton.getWorks()))
  while time.time() < float_maxTime:
   float_maxTime1 = (float_maxTime - time.time()) *.33 + time.time()
   float_maxTime2 = (float_maxTime - time.time()) *.66 + time.time()
   float_maxTime1 = (float_maxTime - time.time()) *.50 + time.time()
   float_maxTime2 = (float_maxTime - time.time()) *1.0 + time.time()
   print 'In outer loop ######'
   print 'about to make cluster set'
   clst_1 = ClusterSet( int_randomClusterCount=int_numberOfClusters )
```

```
     print 'about to perform first clustering'
     clst_1.performClustering( [], 1 )
     print 'about to perform first convergence'
     clst_1.convergeClusters( float_maxTime1, Config.int_minimumConvergenceIterations
)
     lstClus_1 = clst_1.getClusters()
     clst_2 = ClusterSet( int_randomClusterCount=Config.int_createClusterCount )    # A
set of clusters of clusters
     print 'about to perform second clustering'
     clst_2.performClustering( lstClus_1, 0 )        # Make clusters of clusters
     print 'about to merge'
     clst_2.merge()     # Change from clusters of clusters to clusters of works
     print 'about to perform second convergence'
     clst_2.convergeClusters( float_maxTime2, Config.int_minimumConvergenceIterations
)
     clst_2.performClustering( [], 1, bool_binary=0 )
     print 'about to perform third convergence'
     clst_2.convergeClusters( float_maxTime1, Config.int_minimumConvergenceIterations,
bool_binary=0 )
     float_fabulousness = clst_2.getAverageSquaredUniqueArtist()
     print 'max unique:', clst_2.getMaxCountUniqueArtist(), '   min unique:',
clst_2.getMinCountUniqueArtist()
     print ' avg unique:', clst_2.getAverageCountUniqueArtist(),'   fabulousness:',
float_fabulousness
     if float_fabulousness < float_mostFabulous:
       fil_ = open('tuninginfo.txt', 'w')
       fil_.write('float_tuningRankBottom: ' + str( Config.float_tuningRankBottom ) +
'\n')
       fil_.write('float_tuningZeroWeight: ' + str( Config.float_tuningZeroWeight ) +
'\n')
       fil_.write('float_fabulousness: ' + str( float_fabulousness ) + '\n')
       fil_.write('clst_2.getMaxCountUniqueArtist(): ' + str(
clst_2.getMaxCountUniqueArtist() ) + '\n')
       fil_.write('clst_2.getMinCountUniqueArtist(): ' + str(
clst_2.getMinCountUniqueArtist() ) + '\n')
       fil_.write('clst_2.getAverageCountUniqueArtist(): ' + str(
clst_2.getAverageCountUniqueArtist() ) + '\n')

       fil_.close()
       print '###FOUND NEW BEST###'
       print 'writing intermediate'
       float_mostFabulous = float_fabulousness
       clst_2.writeToDisk( 'intermediate.xml' )

       clst_best = clst_2

  elif Config.str_clusterFile:
    clst_cluster = ClusterSet( str_fileName=Config.str_clusterFile )
    clst_cluster.performClustering( Data.singleton.getWorks(), 0 )
    clst_actual.convergeClusters( Config.float_maxTime + time.time(),
Config.int_minimumConvergenceIterations )
  else:
    assert 0, 'Invalid config file option'
  clst_best.writeToDisk( Config.str_outClusterFile )
  print 'done!'
except Exception, str_:
  print 'ERROR'
  print str_
  print '\n\nPress any key to abort:'
  sys.stdin.read(1)
```

Bibliography

Klir, George and Folger, Tina. *Fuzzy Sets, Uncertainty, and Information.* Englewood Cliffs, NJ: Prentice Hall, 1988.

Manly, Bryan F.J. *Multivariate Statistical Methods, A Primer, Second Edition.* London, England: Chapman & Hall, 1994.

Hedges, Larry V. and Olkin, Ingram. *Statistical Methods for Meta-Analysis.* San Diego, CA: Academic Press, 1985.

Snijders, Tom A. B., Maarten Dormaar, Wijbrandt H. van Schuur, Chantal Dijkman-Caes, and Ger Driessen [1990]. "Distribution of Some Similarity Coefficients for Dyadic Banary Data in the Case of Associated Attributes." *Journal of Classification,* 7, pp. 5-31.

http://www.google.com

http://www.interbase.com/

http://www.napster.com